UNIVERSITÀ DI PISA

Computer Engineering

Distributed Systems and Middleware Technologies

"FleetFra Application"

Academic Year: 2024/2025

Authors:

Francesco Taverna

Gabriele Pianigiani

Saverio Mosti

Index

Project Specifications 1

Introduction 1.1

FleetFra is a distributed naval battle game that combines strategic gameplay with modern technologies. It features an **Erlang server** for real-time game logic, a **Spring web server** for business logic and communication, and a **MySQL database** for data storage. The user interface is built with **JavaScript, HTML, and CSS**, providing an intuitive and responsive gaming experience.

Functional requirements 1.2

In this application there are 3 types of actors:

- Unregistered User
- Registered User
- Admin

An Unregistered User can:

- Create an account
- Log-in to an account

A registered User can:

- Log-in/Log-out
- Play a match against another player
- Leave the game

An Admin can:

- Log-in/Log-out
- Browse past games of a certain period of time
- Remove users
- Browse user details

Non-functional requirements 1.3

1. Availability: The application must be able to handle several players gaming at the same time without problems.
2. Coordination: The application must handle the coordination of players in a distributed system
3. Load Balancing: The application must balance the load handled by Erlang servers.
4. Consistency: The application must show the same fields to both players.

Synchronization and Communications issues 1.4

1. Two players exchange attack coordinates with each other during a match. The Erlang Web Server that took the game keeps track of the two fields and marks the coordinates that have been shot. The Erlang server responds to the client whit the result of the action.
2. Each player's game interface must be synchronized with the game. The Javascript client, periodically poll the GameState from the Erlang server, and it updates the GUI.
3. Notify the user if the opponent's turn is over. The Erlang Web Server that took the game coordinates the two users.
4. Notify you and your opponent of the outcome of your action. The Erlang Web Server that took the game coordinates the two users.
5. The server can independently manage any number of games. The Erlang Web Server that took the game creates a process for each game, starts and manages them independently
6. Game states information must be synchronized among Erlang nodes.

Game rules 1.5

The game is turn-based, the player who shoots first is the first to send the request to the server. When the player's turn begins, the GUI will display a notification and the updated battlefield (a kind of matrix) and wait for the player's action. Using the cursor, the player chooses the coordinates in which to fire. The GUI then informs the player of the outcome of the action (Hit or Water). A player's turn ends when they hit the water or if they don't make the move by the timer expires. If the timer expires, no action is sent, and the turn is skipped.

The game ends when all of the ships of one of the two players have been destroyed.

Architecture 2

FleetFra is an online multiplayer naval battle game implemented using Erlang for backend game logic.

The system is designed to handle multiple concurrent games efficiently using a load-balanced architecture with replicated Erlang nodes and WebSocket communication.

Overall System Architecture 2.1

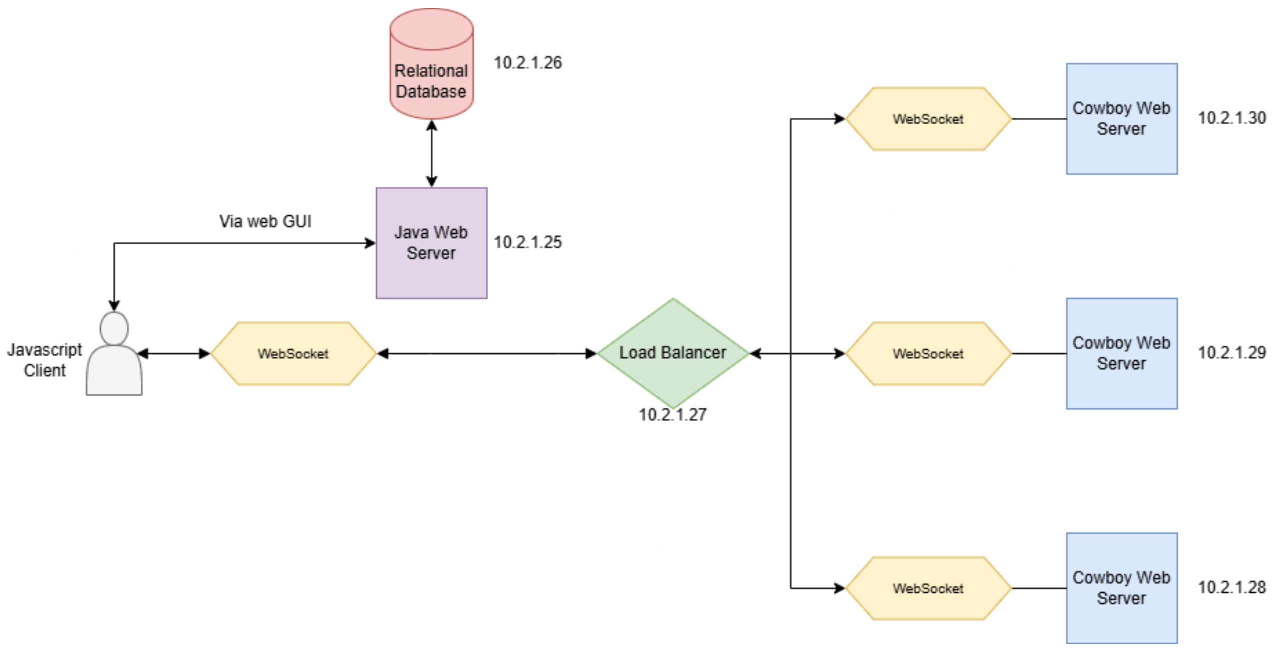


Figure 1 - FleetFra overall architecture.

Erlang Nodes 2.2

The backend consists of three replicated Erlang nodes deployed as Linux containers with the following IPs:

- `10.2.1.30`
- `10.2.1.29`
- `10.2.1.28`

An additional node (`10.2.1.27`) acts as a **NGINX Load Balancer**.

Erlang Web Server 2.2.1

Each Erlang node runs a web server process using **Cowboy** (an open-source Erlang web server).

Why Cowboy?

It has chosen Cowboy because we preferred not to build a web server from scratch; additionally, it is very popular in the **Erlang web server** ecosystem.

Cowboy follows an **event-driven**, **lightweight process-per-connection** model, as opposed to the traditional thread-per-request model. This design ensures efficient handling of multiple concurrent WebSocket connections without the overhead of excessive threading.

After the WebSocket handshake, Cowboy creates a new Erlang process for that WebSocket connection. This process is responsible for managing the ongoing communication between the client and server.

Every message sent over the WebSocket connection is handled by this Erlang process. This means that each WebSocket connection (per client) operates in its own lightweight Erlang process.

Why WebSocket?

It has been decided to use WebSocket for HTTP communication with clients (browsers) because WebSocket supports a persistent connection. Once established, WebSocket connections remain open throughout the session, eliminating the need for repeated HTTP requests.

With a persistent connection, the user experiences significantly less overhead, as they don't need to open a new HTTP connection for each message they send. Additionally, the Erlang node is under much less stress because it doesn't have to open a new WebSocket connection for every request.

It has been estimate that a **FleetFra** match will generate approximately 20 to 30 requests per user during the course of the match. Therefore, the benefits of using WebSocket are evident, as it reduces both client-side and server-side overhead.

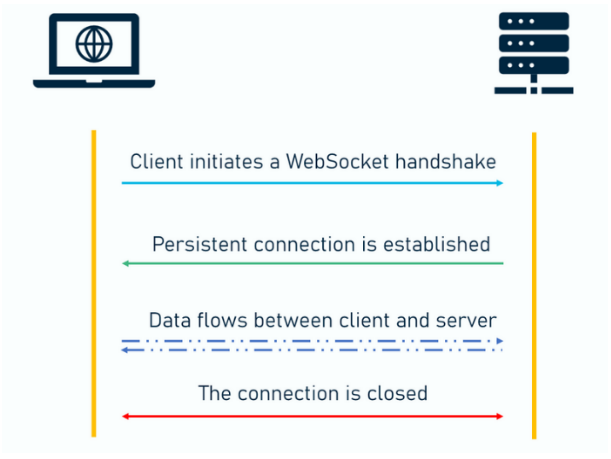

Client initiates a WebSocket handshake
Persistent connection is established
Data flows between client and server
The connection is closed

Figure 2 - How WebSockets works.

Game State Management 2.2.2

Each Erlang node runs a separate process called the **Game State Manager**, implemented as a **gen_server**. This module is responsible for storing the game states of ongoing matches on the node using an **ETS table**.

The Game State Manager is responsible for:

- **Storing** and **providing** access to active game states.
- **Periodically** removing stale (inactive for at least 1 hour) or completed game states to prevent memory leaks.
- **Synchronizing** with other game state managers during write operations.

Eventual consistency: When a Game State Manager receives a request to **put** (insert or update) or **delete** a game state, it propagates the operation to the other two managers asynchronously, leveraging the **gen_server** paradigm.

Below is the data structure stored in the ETS:

```
-record(game,{game_id, players, battlefields, current_turn, game_over, winner, created_at}).
```

The synchronization process ensures fault tolerance, as described in the **NGINX** section of this document.

Load Balancing and Fault Tolerance 2.2.3

To distribute incoming requests evenly across the three Erlang nodes, It has been configured **NGINX** as a load balancer on one of the six nodes in the cluster.

NGINX serves as a **middleware solution** responsible for **efficiently balancing requests** among the three nodes. The balancing mechanism is deterministic, using a **hash function** based on the **GameID** (included in each HTTP request as a GET parameter). This ensures that both players in a match are routed to the same Erlang node, optimizing performance.

However, in rare cases, players might be connected to different Erlang nodes. While this does not cause functional issues, thanks to the **synchronization of the GameState ETS**, it may introduce **performance overhead** due to the synchronization process.

Client Connection and Gameplay Flow 2.2.4

1. **Client 1** establishes a WebSocket connection.
2. **Client 2** establishes a WebSocket connection.
3. **Client 1** sends a `start_game` message containing GameID, PlayerID1, and their battlefield.
4. **Client 2** sends a `start_game` message containing GameID, PlayerID2, and their battlefield.

The first client to send `start_game`, is the one who must send the first `move`.

5. The player whose turn it is sends a move containing **GameID, PlayerID, Row, and Col**.
6. The Erlang server processes the move and responds with the **outcome (Hit/Miss)**.

7. The opponent periodically sends a `get_game_info` request to check the updated state.

The turn alternates, and steps 5-7 repeat until a player wins.

8. When a player wins, the **VICTORY** message is sent, and the WebSocket connections are closed.
9. After a predefined time (1 hour of inactivity), the game's **ETS entry is deleted**.

Interaction Between Components 2.2.4

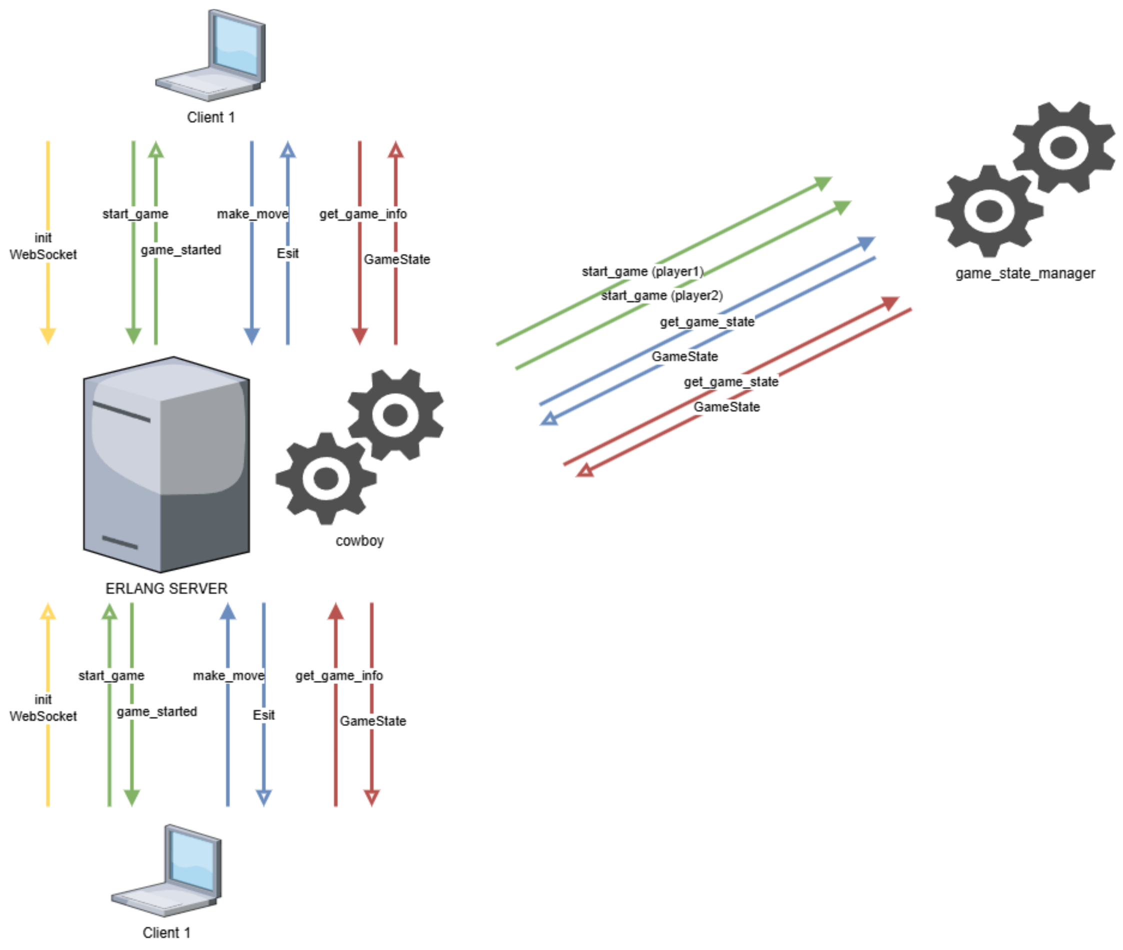


Figure 3 - System components interaction upon each type of operation.

Upon receiving a client request, the Web Server handles the request using the `process_request` function from the `fleetfra_chin_handler` module. This function processes the request by parsing the body and based on the value of the `type_request` field in the JSON (which indicates the type of operation), it determines the appropriate handling approach.

To avoid maintaining an always-on process on the server, the status of a match is stored in an **ETS** (Erlang Term Storage) table, which is managed the `game_state_manager`.

All operations in the ETS are synchronized across the Erlang nodes in the cluster to ensure consistency.

The `start_game` procedure is a two-step process carried out by both clients:

1. When the first `start_game` request is received via WebSocket, a partial GameState is stored in the ETS.

2. Upon receiving the second `start_game` request, the partial GameState is retrieved, completed, and updated in the ETS.

For both `move` and `get_game_info` requests, the GameID (specified in the body of the request) is used to retrieve the associated GameState from the ETS (if it exists) and return it to the client.

In the case of a **move** request, after retrieving the GameState, it is validated. If the move is legitimate, the GameState is modified, and the updated version of the GameState overwrites the old one in the ETS.

Client Game State Retrieval 2.2.5

The Erlang server does not push updates to clients proactively (this could be a potential upgrade for the application).

Game State Retrieval Polling: The non-active player must request the latest state using `get_game_info`, and the response contains a JSON file (approximately 16 Kilobyte) with all the necessary information (the full game state) to update the GUI and keep the client running.

This solution, in the worst case, may result in a **delay of up to 1 second** for the user.

Why did we choose this solution?

Although this solution is somewhat inefficient in terms of bandwidth usage, it was considered the best option for our use case.

The "push approach" is suitable when the minimum response time is a critical requirement (as is often the case when two scripts are communicating).

In our scenario, however, two **human** users are involved, and a (worst-case) 1-second delay did not represent a significant issue. As a result, it has been opted for this **simple** and effective solution.

The only con of this approach remains the bandwidth consumption.

That said, a solution in which the Erlang server itself pushes updates to clients would be superior but more complex to implement.

Spring and MySQL DB 2.3

The Spring Web Server serves as a way to manage various functionalities within the application. It handles user authentication, data retrieval from the MySQL DB, and processing requests from clients. Specifically, its responsibilities include:

- **User Authentication**: Verifying user credentials and granting access to authenticated users.
- **Signup**: Allows users to create new accounts within the application, providing necessary credentials and information for registration.
- **Search random user for a game (Matchmaking)**: Provides users with the ability to find an opponent to play (it binds the player with the first one waiting in the queue).
- **View and remove users (only for admin)**: Grants administrative users the privilege to view all user details and remove users from the platform.
- **Unique ID Assignment**: Assigning a unique identifier to each game session, ensuring distinct identification and management of concurrent games.

Once the two players are ready, the Spring Web Server orchestrates the commencement of the game session by sending the name of the opponent to each player and the game ID to be managed in Javascript.

Matchmaking 2.3.1

In this Spring-based matchmaking system, a concurrent queue (ConcurrentHashMap) is used to manage players looking for a game. When a player sends a matchmaking request via the */game* endpoint, the *UserService* first checks if there is another player already waiting. If a match is found, the second player is paired with the first, and the waiting thread of the first player is notified, allowing the match to proceed. If no match is available, the player is added to the queue and put in a waiting state using *wait()*. When another player arrives, they are matched, and the waiting thread is awakened using *notify()*. The use of synchronized blocks ensures thread safety, preventing race conditions when modifying shared resources.

User Manual 3

Login & Registration 3.1

The start page looks like the following image.

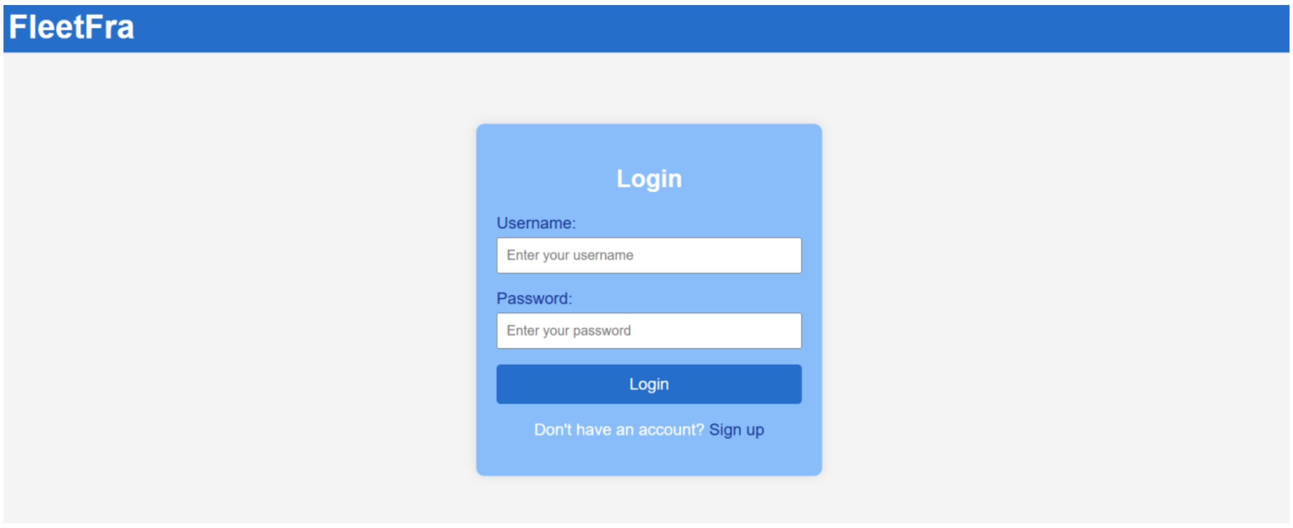


Figure 4 - FleetFra start page.

Login 3.1.1

- Upon launching the application, the user will see a login form where he/she can enter his/her **username** and **password**.
- The user by clicking the **Login** button, access the game.
- If the credentials are incorrect, an **error message** will be displayed.

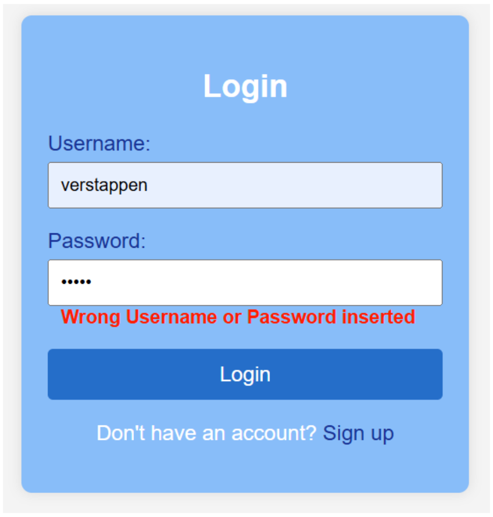
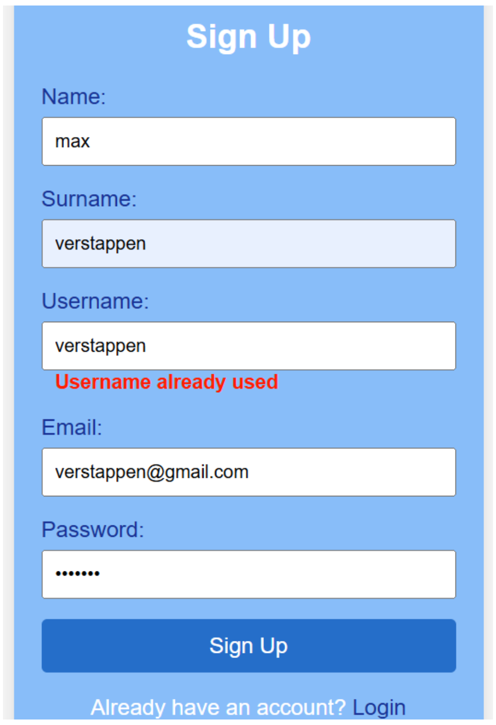

Login
Username:
verstappen
Password:
•••••
Wrong Username or Password inserted
Login
Don't have an account? Sign up

Figure 5 - Error message for wrong credentials during login.

Signup 3.1.2

- Clicking on **Sign Up** the user can switch to the registration form.
- The user can enter the required fields such as **email, username, password, and other necessary details**.

- The user can submit the form to create an account.
- Any invalid input will trigger an **error message**.



Sign Up

Name: max

Surname: verstappen

Username: verstappen

Username already used

Email: verstappen@gmail.com

Password: •••••••

Figure 6 - Signup form with invalid username error message shown.

Game Setup 3.2

1. After logging in, the user will see the battlefield with two grids:
   - **User Grid**: The grid in which the user can position the fleet. The player's username is shown above the left grid.
   - **Opponent Grid**: The right grid where the user will attack.

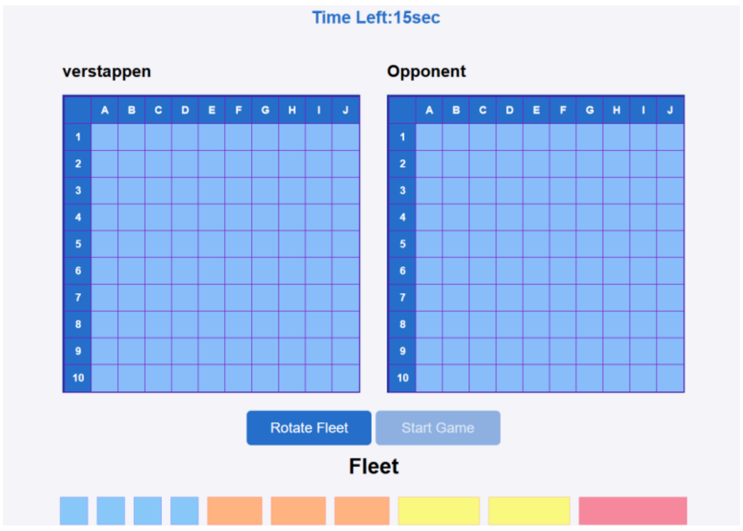


Figure 7 - Initial game page.

2. Below the grids, two buttons are available:
   - **Rotate Fleet:** Allows you to rotate your ships.

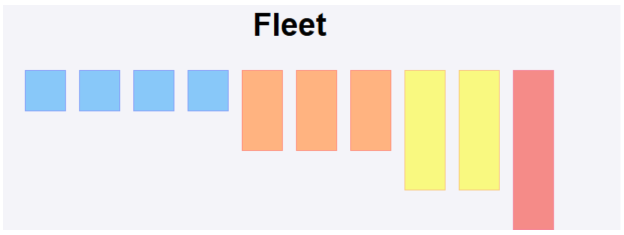


Figure 8 - Fleet after the rotation.

   - **Start Game:** Enabled only when all ships are positioned, it allows to wait for another player to start a match.

3. To insert a ship in the grid, the user can click a ship and then click the cell of the grid in which he/she wants to insert the ship. Once it is inserted, the ship is shown in **green**, and the adjacent cells are shown in **light-blue** and are not available to place a new ship. Before inserting a ship, the cells in which the mouse is pointing to insert the ship are shown in **orange**.

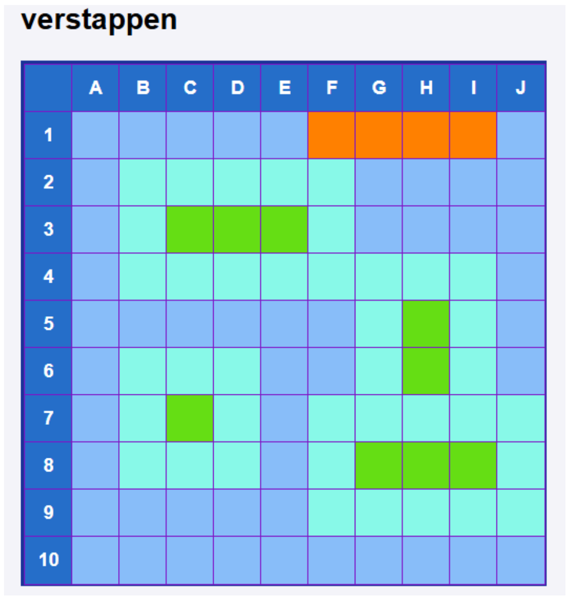


Figure 9 - Player's grid during the ship positioning.

4. The user can click **Start Game** to enter the matchmaking phase.
5. A waiting window will appear while searching for an opponent.

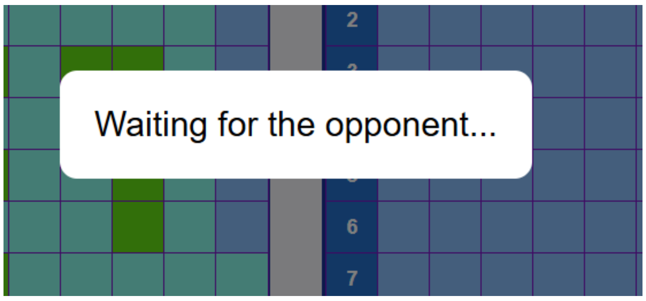

Waiting for the opponent...

Figure 10 - Matchmaking waiting window.

6. Once an opponent is found, a confirmation message appears, and after **2 seconds**, the game begins.

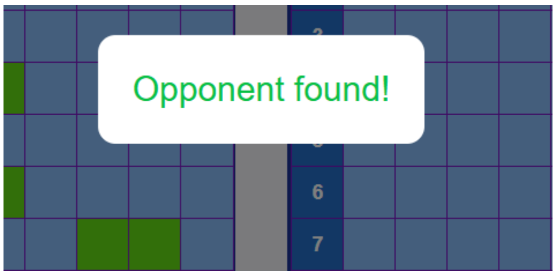
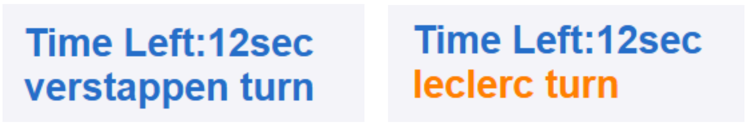

Opponent found!

Figure 11 - Opponent found with the matchmaking.

Gameplay Mechanics 3.3

Turn-Based System 3.3.1

- The username of the opponent will appear over the opponent's grid.
- A **15-second timer** appears at the top, indicating the time left for the move.
- The **active player's username** is displayed. The active player will see the player's turn in **blue**, the waiting player will see the player's turn in **orange**.


Time Left:12sec
verstappen turn

Time Left:12sec
leclerc turn

Figure 12 - Active and waiting player turn.

Making a Move 3.3.2

- The active player can click on a cell in the opponent's grid to attack.
- If he/she hits a ship, the cell turns **red**.
- If he/she hit water, the cell turns light blue.
- A successful hit allows the player to keep the turn and make another move.
- Missing or running out of time ends the player's turn, passing it to the opponent.
- The waiting player can see the moves made by the opponent in his grid.

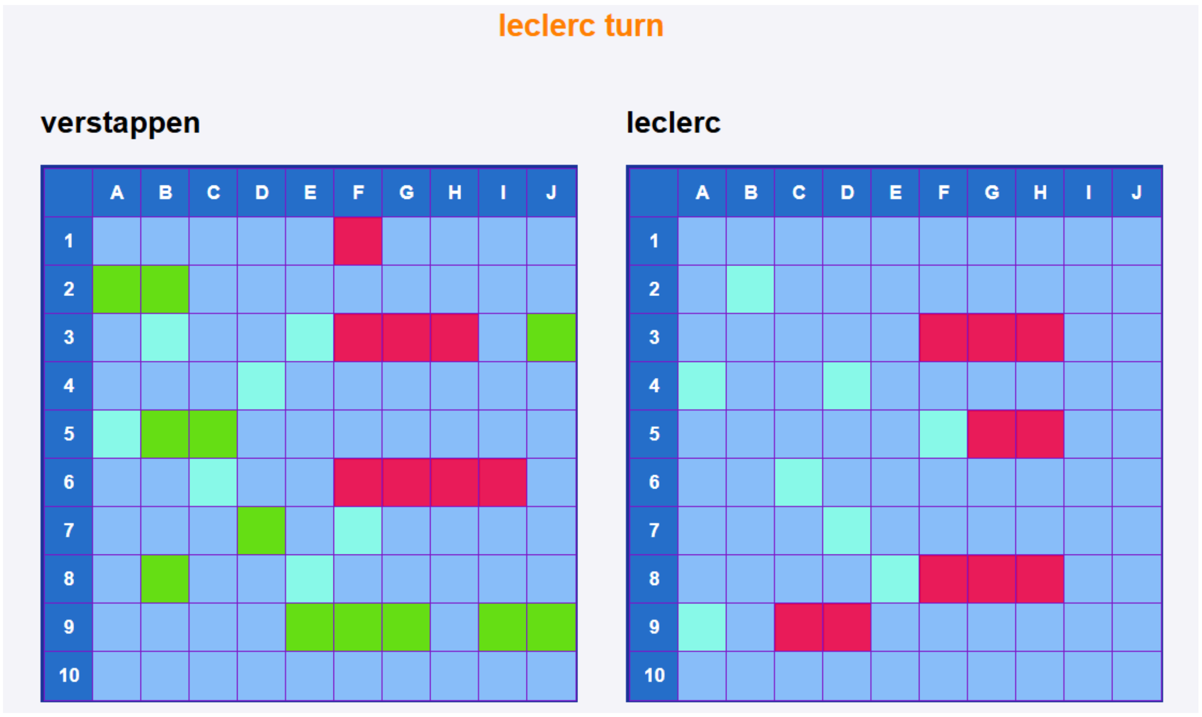


Figure 13 - Player grids during a match.

End of the game 3.3.3

1. When the game ends, a message displaying the result appears for **2 seconds**.

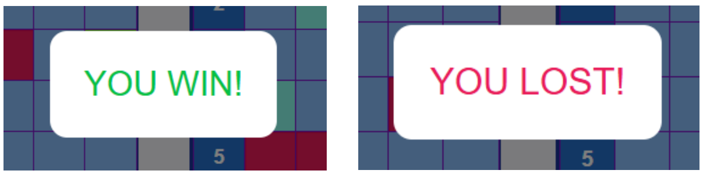


Figure 14 - End of the game for both players.

2. After the match, the page is reloaded, and the user can start a new match by positioning the ships and clicking on **Start Game**.
3. To log out, the user can click the **Logout** button in the top-right corner.

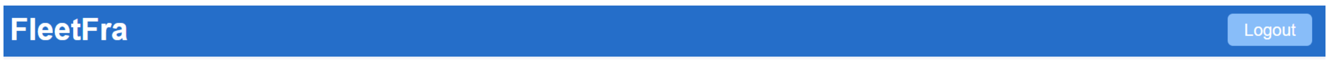


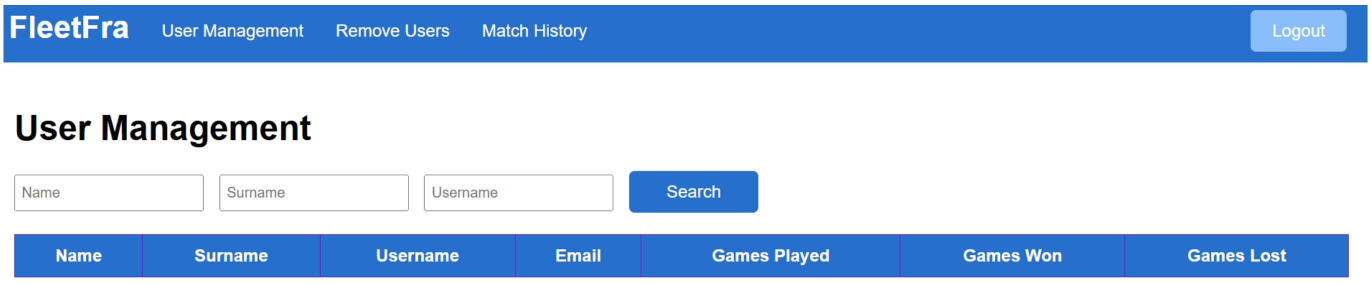
Figure 15 - Top bar of the game page with logout button.

Administrator Guide 3.4

Admin Options 3.4.1

The admin page has a menu bar on the top with four different options:

1. User management.
2. Remove Users.
3. Match history.
4. Logout button.



Name	Surname	Username	Email	Games Played	Games Won	Games Lost

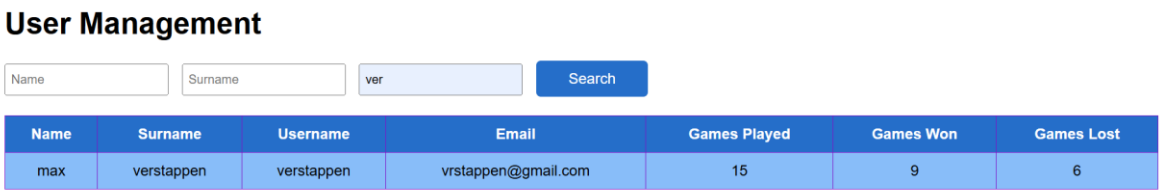
Figure 16 - Home page of the admin.

User Management 3.4.2

The administrator has access to a panel with three input fields:

- **Name**
- **Surname**
- **Username**

1. A button is available to **search** users:
    - If no values are entered, all users are displayed.
    - If specific values are entered, only users matching the criteria are shown.
    - The users are displayed with their personal information and statistics on: **Games Played**, **Games Won**, **Games Lost**.



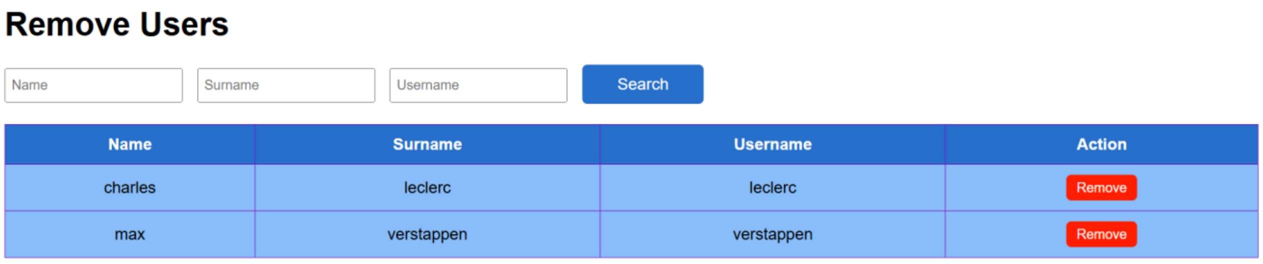
Name	Surname	Username	Email	Games Played	Games Won	Games Lost
max	verstappen	verstappen	vrstappen@gmail.com	15	9	6

Figure 17 - Shown searched users.

Remove Users 3.4.3

- The same input fields are available.
- The table shows information like the name, surname and username of the users.

- A **Remove** button appears in the table to delete a specific user.



Name	Surname	Username	Action
charles	leclerc	leclerc	Remove
max	verstappen	verstappen	Remove

Figure 18 - Shown searched users to remove.

If there are no users that match the search criteria, in both cases is shown an **error message**.

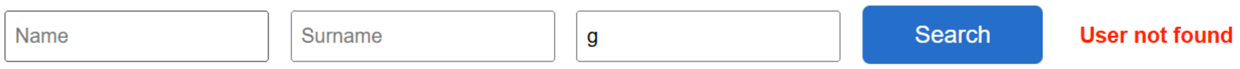


Figure 19 - User not found error.

Match History 3.4.4

- Includes two date input fields allow selecting a start and end date.
- Clicking the button without selecting dates shows an **error message**.

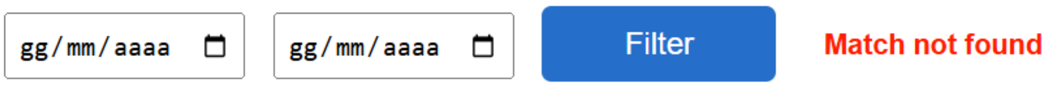
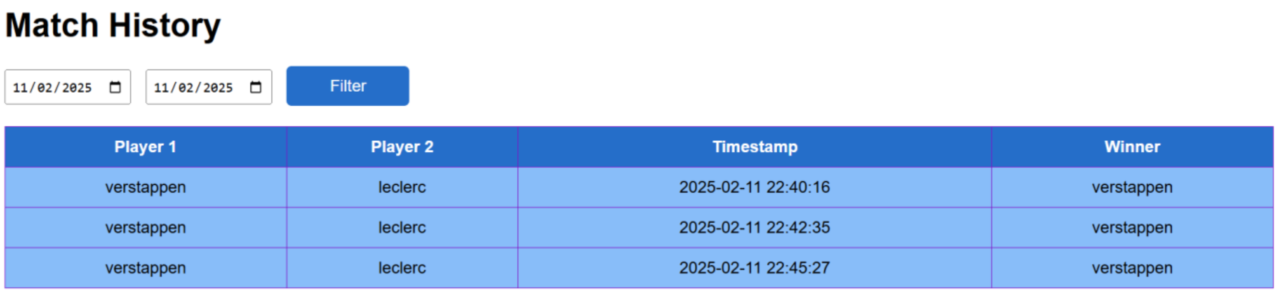


Figure 20 - Match not found error.

- If valid dates are entered, matches played within the range are displayed.



Player 1	Player 2	Timestamp	Winner
verstappen	leclerc	2025-02-11 22:40:16	verstappen
verstappen	leclerc	2025-02-11 22:42:35	verstappen
verstappen	leclerc	2025-02-11 22:45:27	verstappen

Figure 21 - Match played on a certain date.